CHAPTER 3: CONCURRENT PROCESSES AND PROGRAMMING

Chapter outline

- Thread implementations
- Process models
- The client/server model
- Time services
- Language constructs for synchronization
- Concurrent programming systems

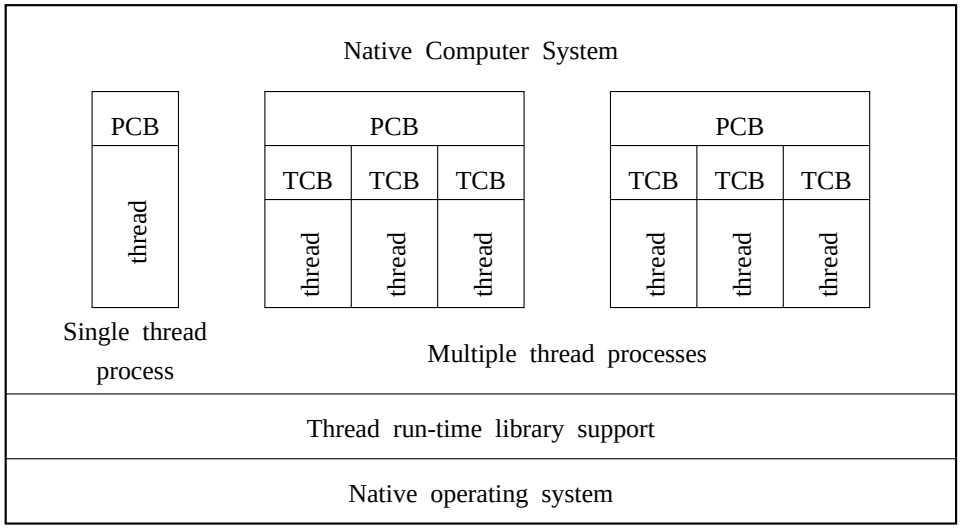
Processes and threads

- *Processes*: separate logical address space
- *Threads*: common logical address space

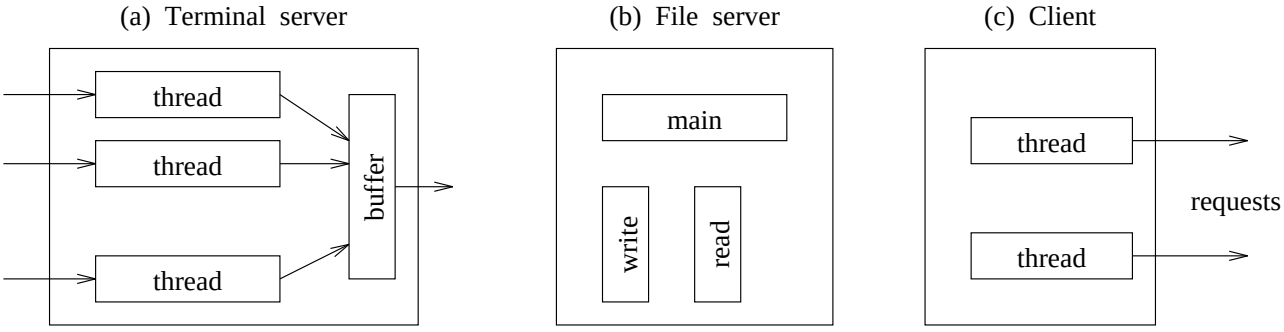
Major Issues

- Process/thread creation
- Light weight context switching
- Blocking and scheduling

Two-level concurrency of processes and threads



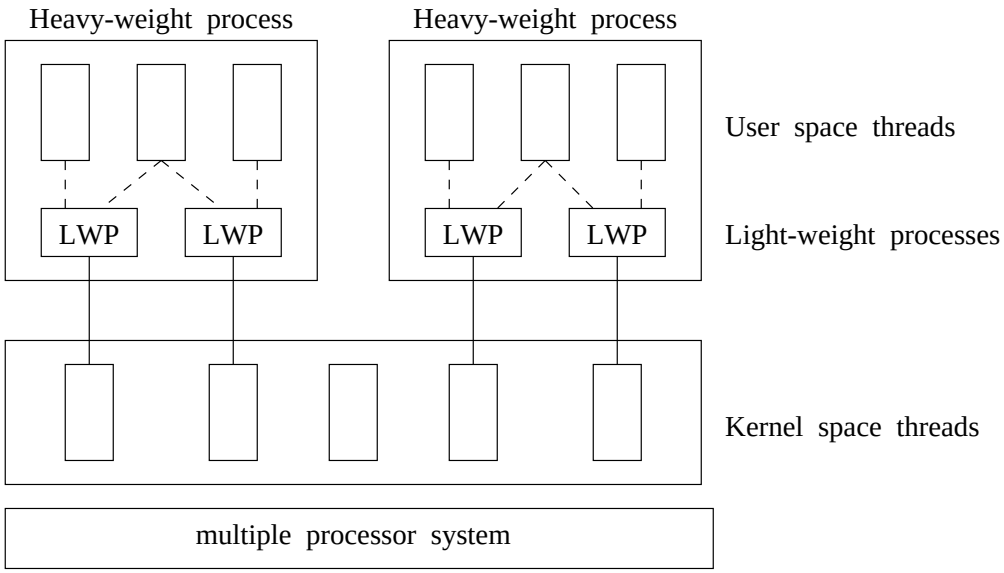
Thread applications



Thread implementations

- *User space*: simple but non-preemptable
- *Kernel space*: efficient but not portable

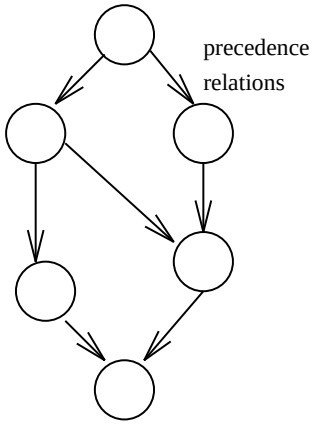
Solaris thread implementation



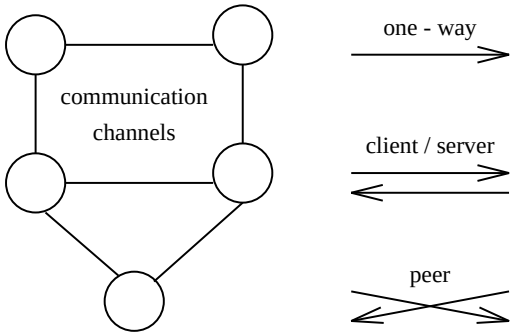
Process models

Synchronous Process, Asynchronous Communication, Time-Space

Graph representations

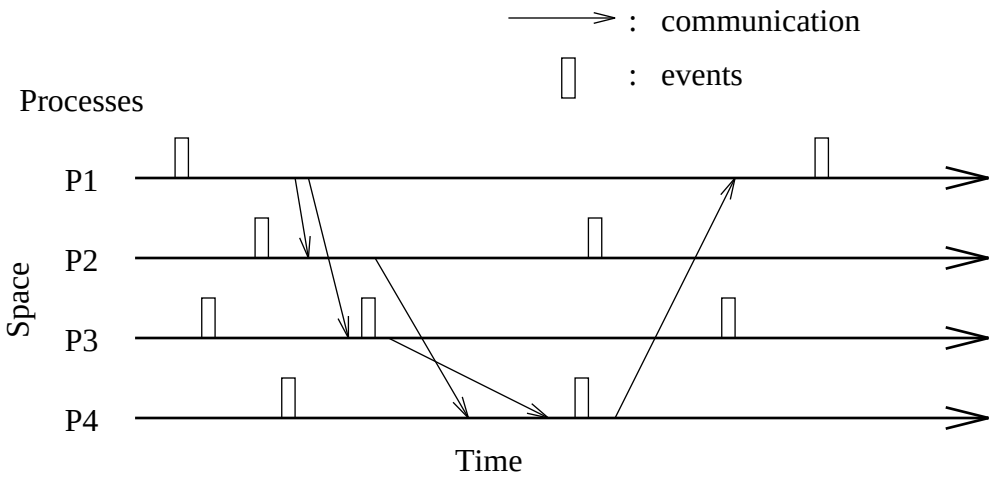


Synchronous process graph

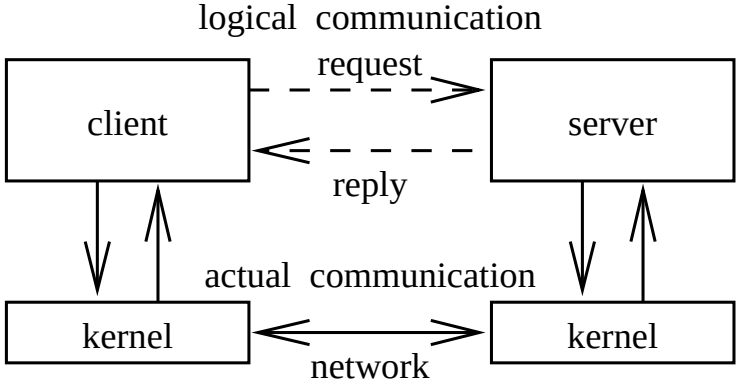


Asynchronous process graph and communication scenarios

Time-space model



Client/server model

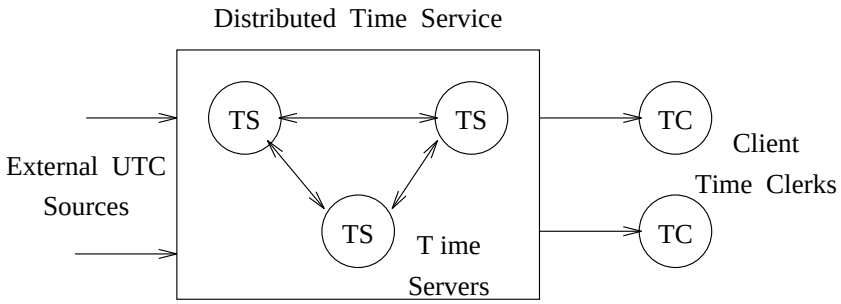


Time services

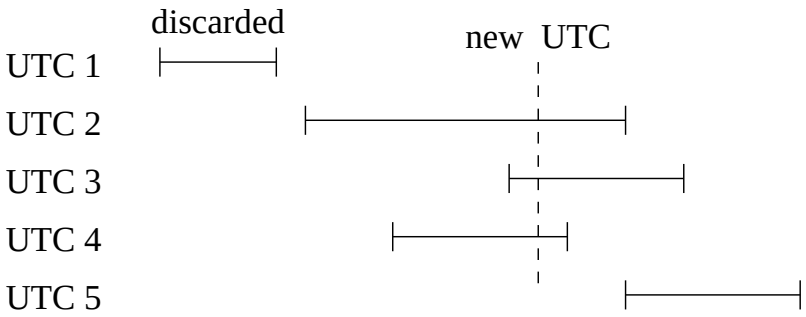
- time and timer
- physical and logical clocks

Physical clock

A distributed time service architecture



Time Discrepancies



Lamport Logical Clock

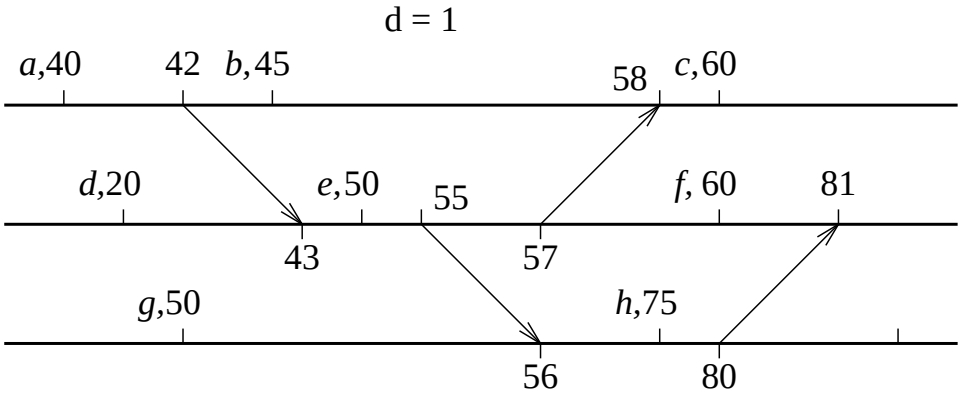
The *happens-before* relationship: $\rightarrow$

1. If $a \rightarrow b$ within a same process then $C(a) < C(b)$.
2. If a is the sending event of P_i and
   b is the corresponding receiving event of P_j, then $a \rightarrow b$ and
   $C_i(a) < C_j(b)$.

For it to be possible for a to have an influence on b, then $a \rightarrow b$ must be true.

Implementation:

$C(b) = C(a) + d$ and
$C_j(b) = max(TS_a + d, C_j(b))$,
where TS_a is the timestamp of the sending event and d is a positive number.



So, $a \rightarrow b \implies C(a) < C(b)$, but $C(a) < C(b) \not\Longrightarrow \quad a \rightarrow b$.

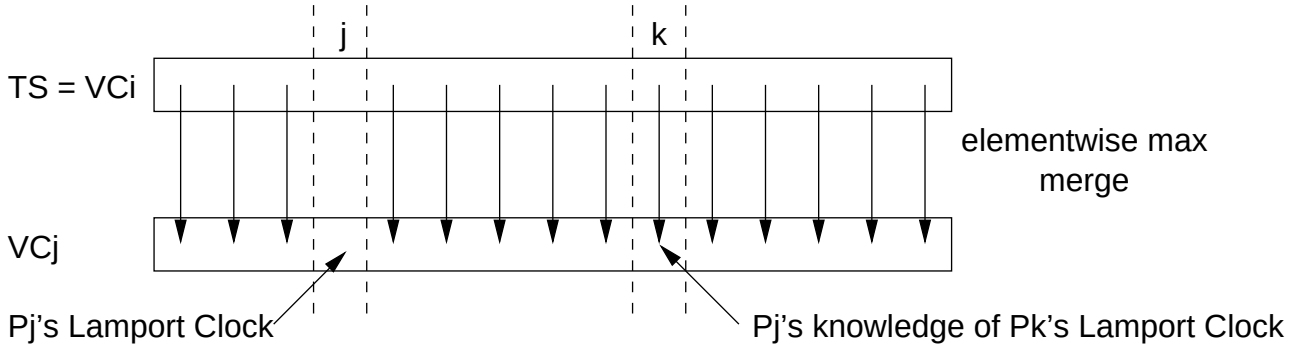
Vector Logical Clock

Used so that if $C_i(a) < C_j(b)$ then $a \rightarrow b$.

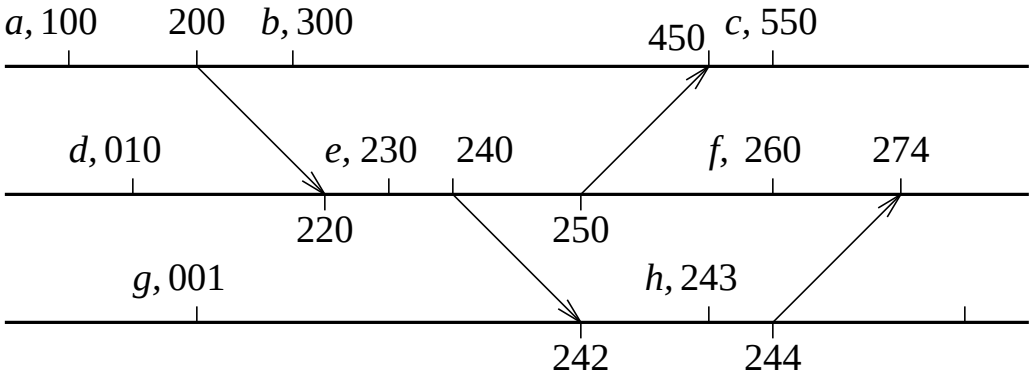
Define $VC_i = [TS_1, TS_2, ..., C_i, ..., TS_n]$,
where n is the number of cooperating processes.

On message receipt, use *pair-wise maximum*.

$VC_j[j] = VC_j[j] + d$
$VC_j[k] = \max(VC_j[k], TS_i[k]) \quad : \; l = 1..n$



$VC_j[j]$ is P_j's count of events that have occured at P_j,
$VC_j[k]$ is P_j's knowledge of events that have occured at P_k.



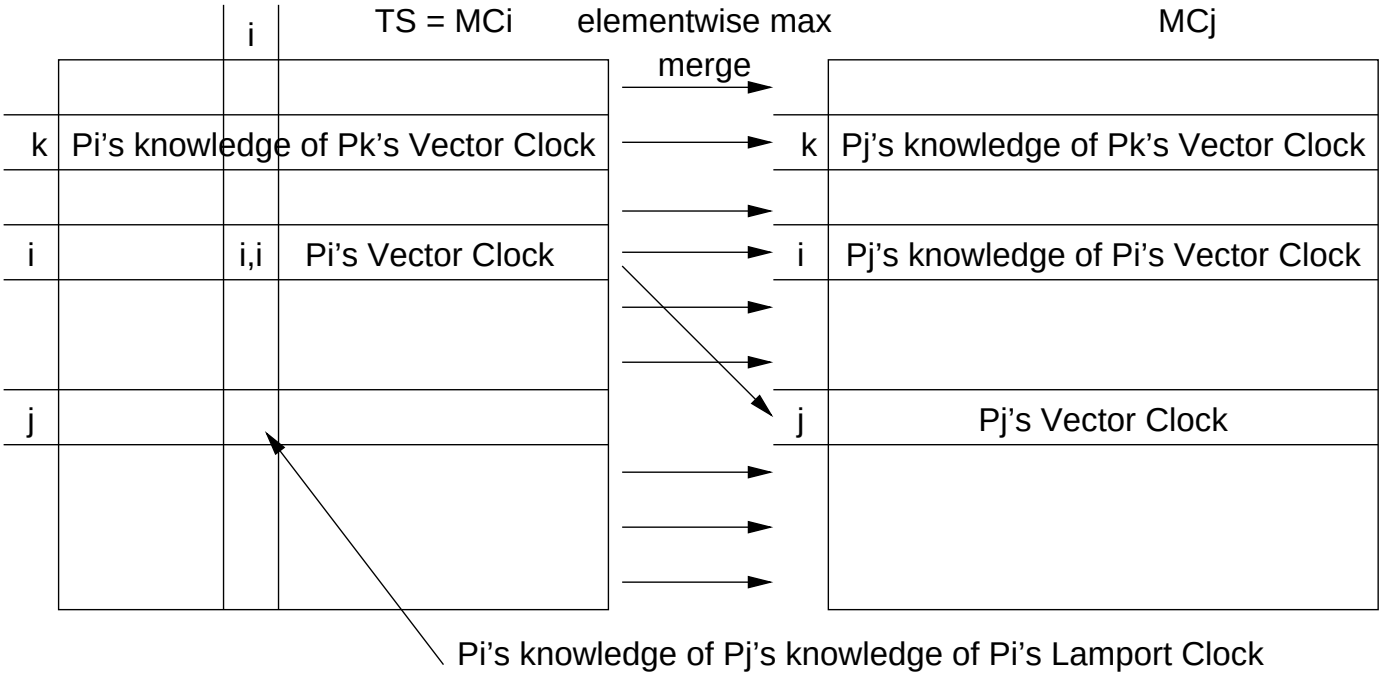
Matrix Logical Clock

MC_i represents
P_i's knowledge of its local events ($MC_i[i,i]$),
its knowledge of the events that P_j knows about ($MC_i[i,j]$), and
its knowledge of P_j's knowledge of events at P_k ($MC_i[j,k]$).

$MC_i[i,i] = MC_i[i,i] + d$ – P_i updates local event counter on send

When P_j receives a message from P_i with timestamp TS, $MC_j[j,l] = \max(MC_j[j,l], TS_i[i,l])$: $l = 1..n$ update vector clock, and

$MC_j[k,l] = \max(MC_j[k,l], TS_i[k,l])$: $k = 1..n$, $l = 1..n$ update P_k's knowledge of P_l's counter



Concurrent languages

- Specification of concurrent activities
- Synchronization of processes
- Interprocess communication
- Nonderterministic execution of processes

Language constructs

- Program structure
- Data structure
- Control structure
- Procedure and system call
- Input and output
- Assignment

Synchronization mechanisms and language facilities

Synchronization Methods	*Language Facilities*
Shared-Variable Synchronization	
semaphore	shared variable and system call
monitor	data type abstraction
conditional critical region	control structure
serializer	data type and control structure
path expression	data type and program structure
Message Passing Synchronization	
communicating sequential processes	input and output
remote procedure call	procedure call
rendezvous	procedure call and communication

Shared-variable synchronization

- *Semaphore* and *conditional critical region*
- *Monitor* and *serializer*
- *Path expression*

Classic Problems

- Critical Section
- Dining Philosophers
- Readers/Writers
- Producer-Consumer

Example: the Reader/Writer Problems
synchronization + concurrency

Basics

- if DB empty, allow anyone in
- if reader in DB, writer not allowed in
- if writer in DB, nobody allowed in

Lock Requested	Lock Held	
	Read Lock	Write Lock
Read Lock	✓	✗
Write Lock	✗	✗

Variations

- *reader preference*
  Allow a reader in if other readers are in
- *strong reader preference*
  Allow readers in when writer leaves
- *weak reader preference*
  When writer leaves, select a process at random
- *weaker reader preference*
  Allow a writer in when writer leaves
- *writer preference*
  Do not allow readers in if writer is waiting

Semaphore solution to the weak reader preference problem

var mutex=1, db=1: **semaphore**; rc=0: **integer**

reader processes	**writer processes**
do (forever)	do (forever)
BEGIN	BEGIN
otherStuff()	otherStuff()
P(mutex)	
rc := rc + 1	
if rc = 1 **then P**(db)	**P**(db)
V(mutex)	
read database	write database
P(mutex)	
rc := rc -1	
if rc = 0 **then V**(db)	**V**(db)
V(mutex)	
END	END

Monitor solution

```
rw : monitor
var rc : integer; busy : boolean; toread, towrite : condition;

procedure startread                 procedure endread
begin                               begin
if busy then toread.wait;
rc := rc + 1;                       rc := rc - 1;
toread.signal;                      if rc = 0 then towrite.signal;
end                                 end

procedure startwrite                procedure endwrite
begin                               begin
if busy or rc ≠ 0
then towrite.wait;                  busy := false;
busy := true;                       toread.signal or towrite.signal;
end                                 end

begin rc := 0; busy := false end
```

```
reader processes                    writer processes

do (forever) BEGIN                  do (forever) BEGIN

otherStuff()                        otherStuff()

rw.startread                        rw.startwrite
read database                       write database
rw.endread                          rw.endwrite

END                                 END
```

CCR solution

```
var db: shared; rc: integer;

reader processes                              writer processes

region db begin rc := rc + 1 end;             region db when rc = 0
read database                                 begin write dat abase end
region db begin rc := rc - 1 end;
```

Serializer solution

```
rw : serializer
var readq, writeq: queue; rcrowd, wcrowd: crowd;

procedure read
begin
enqueue(readq) until empty(wcrowd);
joincrowd(rcrowd) then begin read database end;
end

procedure write
begin
enqueue(writeq) until (empty(wcrowd) and empty(rcrowd));
joincrowd(wcrowd) then begin write database end;
end
```

Path Expression solution

```
path 1:([read],write) end
```

Message Passing Synchronization

- *Asynchronous*: non-blocking send, blocking receive
- *Synchronous*: blocking send, blocking receive

Mutual exclusion using asyn. msg. passing

process P_i	**channel server**	**process** P_j
begin	**begin**	**begin**
receive(channel)	create channel	receive(channel)
critical section	send(channel)	critical section
send(channel)	manage channel	send(channel)
end	**end**	**end**

Mutual exclusion using syn. msg. passing

process P_i	**semaphore server**	**process** P_j
begin	**loop**	**begin**
send(sem,msg)	receive(pid,msg)	send(sem,msg)
critical section	send(pid,msg)	critical section
receive(sem,msg)	**end**	receive(sem,msg)
end		**end**

Communicating Sequential Processes (CSP)

P: $Q!exp$, Q: $P?var$, and *guarded* commands
Process P executes $Q!(x + y)$,
. then expression $x + y$ is evaluated and sent to process Q.
Process Q executes $P?z$,
. then process Q sets variable z to the value received from process P

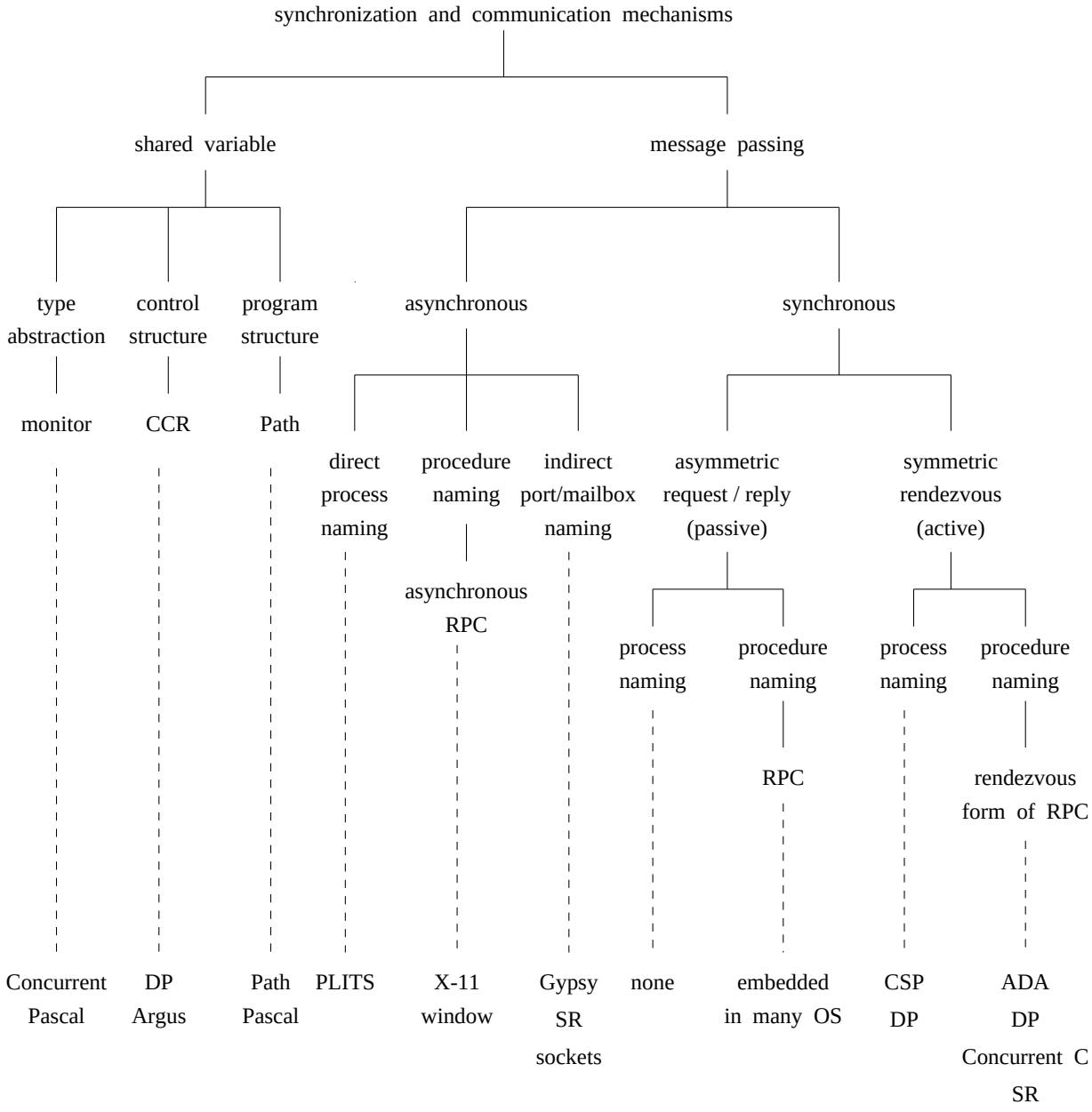
ADA rendezvous

```
task rw is
    entry startread;
    entry endread;
    entry startwrite;
    entry endwrite;
end

task body rw is
    rc: integer := 0;
    busy: boolean := false;
begin
loop
    select
        when busy = false →
        accept startread do rc := rc + 1 end;
    or
        →
        accept endread do rc := rc - 1 end;
    or
        when rc = 0 and busy = false →
        accept startwrite do busy = true end;
    or
        →
        accept endwrite do busy = false end;
end loop
end;
```

Concurrent Programming Languages

A taxonomy



Coordination languages

- *OCCAM*: based on CSP process model, use PAR, ALT, and SEQ constructors, use explict global links for communication.
- *SR*: based on resource (object) model, use synchronous CALL and asynchronous SEND and rendezvous IN, use *capability* for channel naming.
- *LINDA*: based on distributed data structure model, use tuples to represent both process and object, use blocking IN and RD and non-blocking OUT for communication.

	System	Object model	Channel naming
OCCAM	concurrent programming language	processes	static global channels
SR	concurrent programming language	resources	dynamic capabilities
LINDA	concurrent programming paradigm	distributed data structures	associative tags

Distributed and Network Programming

Programming languages for loosely coupled systems:

ORCA

fork process-name(parameters) [**on** (processor-number)];

operation op(parameters)
guard condition **do** statements;
guard condition **do** statements;

invoke(object, operation, parameters)

$t[1] = 6, A, 8$
$t[6] = 0, B, 0$
$t[8] = 0, C, 0$

JAVA

- Well-defined standard interfaces for integrating software modules
- Capability of running software modules on any machine
- Infrastructure for coordinating and transporting software modules

Applet and system security